# Luminoth Documentation

*Release 0.2.4.dev*

Tryolabs

Nov 17, 2018

Usage:

Luminoth is an open source toolkit for computer vision. Currently, we support object detection, but we are aiming for much more. It's built in Python, using TensorFlow.

The code is open source and available on GitHub.

# CHAPTER 1

# Documentation

## 1.1 Installation

### 1.1.1 Before you start

#### TensorFlow

To use Luminoth, TensorFlow must be installed beforehand.

If you want **GPU support**, you should install the GPU version of TensorFlow with `pip install tensorflow-gpu`, or else you can use the CPU version using `pip install tensorflow`.

You can see more details of how to install TensorFlow manually here, including how to use CUDA and cuDNN.

#### FFmpeg

Luminoth leverages FFmpeg in order to support running predictions on videos. If you plan to use Luminoth with this end, FFmpeg should be installed as a system dependency.

### 1.1.2 Installing from PyPI

Use `pip` to install Luminoth, by running the following command:

```
pip install luminoth
```

#### Google Cloud

If you wish to train using **Google Cloud ML Engine**, the optional dependencies must be installed:

```
$ pip install luminoth[gcloud]
```

### 1.1.3 Installing from source

Start by cloning the Luminoth repository:

```
git clone https://github.com/tryolabs/luminoth.git
```

Then install the library by running:

```
cd luminoth
pip install -e .
```

## 1.2 Getting started

After going through the installation process (see *Installation*), the `lumi` CLI tool should be at your disposal. This tool is the main way to interact with Luminoth, allowing you to train new models, evaluate them, use them for predictions, manage your checkpoints and more. Running it will provide additional information:

```
Usage: lumi [OPTIONS] COMMAND [ARGS]...

Options:
  -h, --help  Show this message and exit.

Commands:
  checkpoint  Groups of commands to manage checkpoints
  cloud       Groups of commands to train models in the...
  dataset     Groups of commands to manage datasets
  eval        Evaluate trained (or training) models
  predict     Obtain a model's predictions.
  server      Groups of commands to serve models
  train       Train models
```

We'll start by downloading a checkpoint. Luminoth provides already-trained models so you can run predictions and get reasonable results in no time (and eventually be able to use them for fine-tuning). In order to access these checkpoints, we first need to download the remote index with the available models.

Checkpoint management is handled by the `lumi checkpoint` subcommand. Run the following to both retrieve and list the existing checkpoints:

```
$ lumi checkpoint refresh
Retrieving remote index... done.
2 new remote checkpoints added.
$ lumi checkpoint list
================================================================================
|           id |                  name |       alias | source |         status |
================================================================================
| 48ed2350f5b2 |   Faster R-CNN w/COCO |    accurate | remote | NOT_DOWNLOADED |
| e3256ffb7e29 |      SSD w/Pascal VOC |        fast |  local | NOT_DOWNLOADED |
================================================================================
```

Two checkpoints are present:

- **Faster R-CNN w/COCO** (48ed2350f5b2): object detection model trained on the Faster R-CNN model using the COCO dataset. Aliased as `accurate`, as it's the slower but more accurate detection model.
- **SSD w/Pascal VOC** (e3256ffb7e29): object detection model trained on the Single Shot Multibox Detector (SSD) model using the Pascal dataset. Aliased as `fast`, as it's the faster but less accurate detection model.

Additional commands are available for managing checkpoints, including inspection and modification of checkpoints (see *Checkpoint management*). For now, we'll download a checkpoint and use it:

```
$ lumi checkpoint download 48ed2350f5b2
Downloading checkpoint...  [####################################]  100%
Importing checkpoint... done.
Checkpoint imported successfully.
```

Once the checkpoint is downloaded, it can be used for predictions. There are currently two ways to do this:

- Using the CLI tool and passing it either images or videos. This will output a JSON with the results and optionally draw the bounding boxes of the detections in the image.
- Using the web app provided for testing purposes. This will start a web server that, when connected, allows you to upload the image. Also useful to run on a remote GPU. (Note, however, that using Luminoth through the web interface is **not** production-ready and will not scale.)

Let's start with the first, by running it on an image aptly named `image.png`:

```
$ lumi predict image.png
Found 1 files to predict.
Neither checkpoint not config specified, assuming `accurate`.
Predicting image.jpg... done.
{
  "file": "image.jpg",
  "objects": [
    {"bbox": [294, 231, 468, 536], "label": "person", "prob": 0.9997},
    {"bbox": [494, 289, 578, 439], "label": "person", "prob": 0.9971},
    {"bbox": [727, 303, 800, 465], "label": "person", "prob": 0.997},
    {"bbox": [555, 315, 652, 560], "label": "person", "prob": 0.9965},
    {"bbox": [569, 425, 636, 600], "label": "bicycle", "prob": 0.9934},
    {"bbox": [326, 410, 426, 582], "label": "bicycle", "prob": 0.9933},
    {"bbox": [744, 380, 784, 482], "label": "bicycle", "prob": 0.9334},
    {"bbox": [506, 360, 565, 480], "label": "bicycle", "prob": 0.8724},
    {"bbox": [848, 319, 858, 342], "label": "person", "prob": 0.8142},
    {"bbox": [534, 298, 633, 473], "label": "person", "prob": 0.4089}
  ]
}
```

You can further specify the checkpoint to use (by using the `--checkpoint` option), as well as indicating the minimum score to allow for bounding boxes (too low will detect noise, too high and won't detect anything), the number of detections, and so on.

The second variant is even easier to use, just run the following command and go to http://127.0.0.1:5000:

```
$ lumi server web
Neither checkpoint not config specified, assuming `accurate`.
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

In there, you'll be able to upload an image and see the results.

And that's it for the basics! Next steps would be:

- Prepare your own dataset to be consumed by Luminoth (see *Adapting a dataset*).

- Train a custom model with your own data, either locally or in Google Cloud (see *Training your own model*).
- Turn your custom model into a checkpoint for easier sharing and usage (see *Working with checkpoints*).
- Use the Python API to call Luminoth models within Python.

## 1.3 Tutorial: real world object detection with Luminoth

In this tutorial, we will learn the workings of *Luminoth* by using it in practice to solve a real world object detection problem.

As our case study, we will be building a model able to recognize cars, pedestrians, and other objects which a self-driving car would need to detect in order to properly function. We will have our model ready for that and see it how to apply it to images and video. We will not, however, add any tracking capabilities.

To follow along easier and not invest many hours each time we want to run the training process, we will build a small toy dataset and show how things go from there, giving tips on the things you need to look at when training a model with a larger dataset.

First, check the *Installation* section and make sure you have a working install.

### 1.3.1 First steps

In this section, we are going to walk through the baby steps of using Luminoth.

The first thing is being familiarized with the Luminoth CLI tool, that is, the tool that you interact with using the `lumi` command. This is the main gate to Luminoth, allowing you to train new models, evaluate them, use them for predictions, manage your checkpoints and more.

To start, you should fetch a couple of images/videos from the internet. We will try to play around with traffic-related stuff (cars, pedestrians, bicycles, etc), so we want images that relate to what you would see on the street. To make it easier, you can use the following free images:

```
https://pixabay.com/en/bicycling-riding-bike-riding-1160860/
https://pixabay.com/en/traffic-rush-hour-rush-hour-urban-843309/
https://pixabay.com/en/grandparents-grandmother-people-1969824/
https://pixabay.com/en/budapest-chain-bridge-jam-pkw-bus-1330977/
https://videos.pexels.com/videos/people-walking-by-on-a-sidewalk-854100
https://videos.pexels.com/videos/people-walking-on-sidewalk-992628
```

But of course, you can Google and try it out with your own images!

#### Using the shell: detecting objects in an image or video

Fire up the shell and go to the directory where your images are located. Let's say we want Luminoth to predict the objects present in one of these pictures (`bicycling-1160860_1280.jpg`). The way to do that is by running the following command:

```
lumi predict bicycling-1160860_1280.jpg
```

You will see the following output:

```
Found 1 files to predict.
Neither checkpoint not config specified, assuming `accurate`.
Checkpoint not found. Check remote repository? [y/N]:
```

What happens is that you didn't tell Luminoth what an "object" is for you, nor have taught it how to recognize said objects.

One way to do this is to use a **pre-trained model** that has been trained to detect popular types of objects. For example, it can be a model trained with COCO dataset or Pascal VOC. Moreover, each pre-trained model might be associated with a different algorithm. This is what **checkpoints** are: they correspond to the weights of a particular model (Faster R-CNN or SSD), trained with a particular dataset.

The case of "accurate" is just a label for a particular Deep Learning model underneath, in this case, Faster R-CNN, trained with images from the COCO dataset. The idea is that Luminoth assumes that by default you want the most accurate predictions, and it will use the most accurate model that it knows about. At this time, it is Faster R-CNN, but that could be replaced in the future and you, as a user, wouldn't have to change your code.

Type 'y' and Luminoth will check the remote index, to see what checkpoints are available. Luminoth currently hosts pre-trained checkpoints for Faster R-CNN (COCO) and SSD (Pascal VOC), though more will be added.

Type 'y' again after it prompts you to download the checkpoint. The checkpoints will be stored in `~/.luminoth` folder.

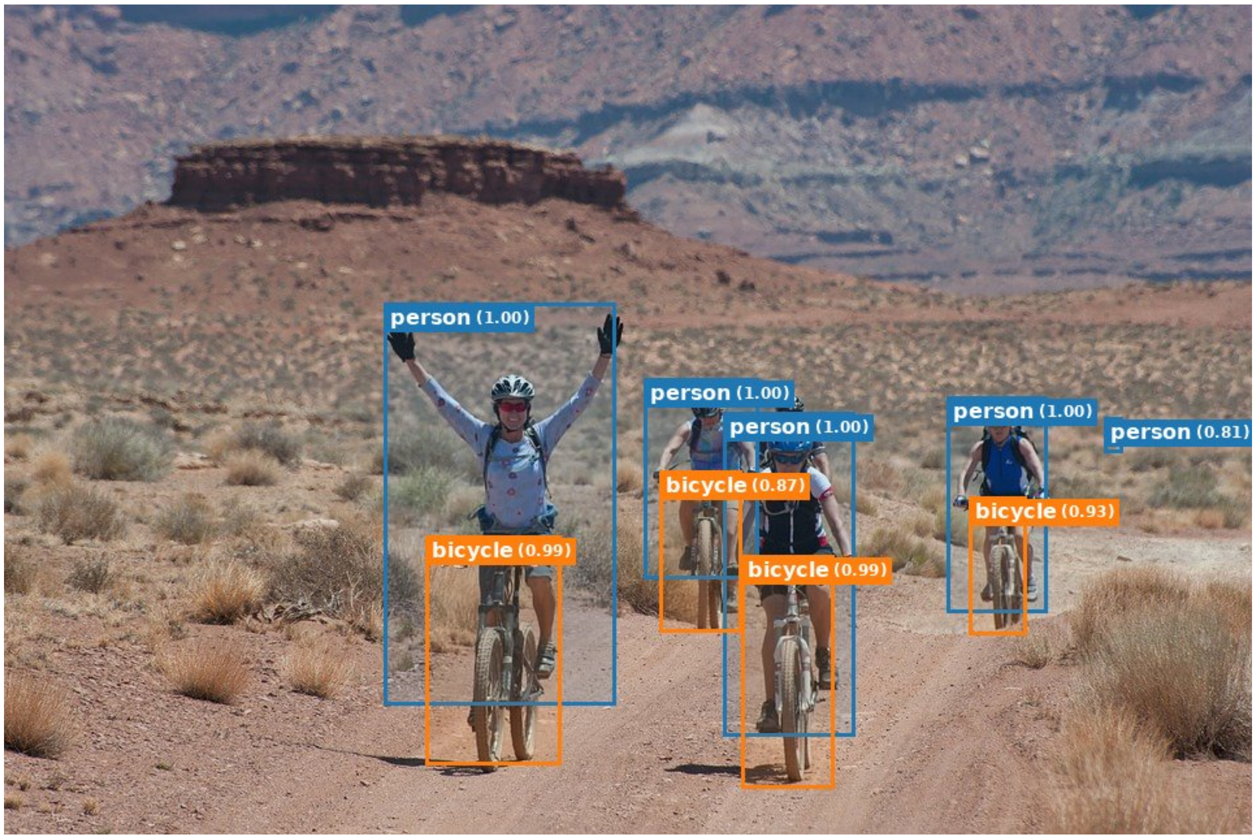
After the download finishes, you will get the predictions for your image in JSON format in the standard output:

```
Predicting bicycling-1160860_1280.jpg... done.
{"file": "bicycling-1160860_1280.jpg", "objects": [{"bbox": [393, 300, 631, 748],
↪"label": "person", "prob": 0.9996}, {"bbox": [978, 403, 1074, 608], "label": "person
↪", "prob": 0.9965}, {"bbox": [670, 382, 775, 596], "label": "person", "prob": 0.
↪9949}, {"bbox": [746, 421, 877, 743], "label": "person", "prob": 0.9947}, {"bbox":␣
↪[431, 517, 575, 776], "label": "bicycle", "prob": 0.9876}, {"bbox": [775, 561, 860,␣
↪792], "label": "bicycle", "prob": 0.9775}, {"bbox": [986, 499, 1057, 636], "label":
↪"bicycle", "prob": 0.9547}, {"bbox": [1135, 420, 1148, 451], "label": "person",
↪"prob": 0.8286}, {"bbox": [683, 480, 756, 621], "label": "bicycle", "prob": 0.7845},
↪ {"bbox": [772, 394, 853, 478], "label": "person", "prob": 0.6044}, {"bbox": [384,␣
↪318, 424, 365], "label": "baseball glove", "prob": 0.6037}, {"bbox": [700, 412, 756,
↪ 471], "label": "backpack", "prob": 0.5078}, {"bbox": [606, 311, 637, 353], "label
↪": "baseball glove", "prob": 0.5066}]}
```

This is probably unintelligible to you, and also not apt for machine consumption since it's mixed with other things in the standard output. However, it's also possible to get the JSON file with the objects plus the actual image with the overlaid bounding boxes. With these commands we can output everything to a `preds` directory:

```
mkdir preds
lumi predict bicycling-1160860_1280.jpg -f preds/objects.json -d preds/
```

If you fetch the resulting image, it should look like this:



Not bad!

You can also run predictions on a **video file** in the exact same way. Note that this is basically independent frame by frame predictions, and has no tracking or interpolation. Try it out! Depending on the length of the video, it can take a while :)

## Exploring pre-trained checkpoints

Whenever you wish to work with checkpoints, you must first run the `lumi checkpoint refresh` command, so Luminoth knows about the checkpoints that it has available for download. The remote index can be updated periodically.

After refreshing the local index, you can list the available checkpoints running `lumi checkpoint list`:

```
================================================================================
|           id |                   name |       alias | source |         status |
================================================================================
| e1c2565b51e9 |    Faster R-CNN w/COCO |    accurate | remote |     DOWNLOADED |
| aad6912e94d9 |      SSD w/Pascal VOC |        fast | remote | NOT_DOWNLOADED |
================================================================================
```

Here, you can see the "accurate" checkpoint that we have used for our predictions before, and that we also have another "fast" checkpoint that is the SSD model trained with Pascal VOC dataset. Let's get some information about the "accurate" checkpoint: `lumi checkpoint info e1c2565b51e9` or `lumi checkpoint info accurate`

```
Faster R-CNN w/COCO (e1c2565b51e9, accurate)
Base Faster R-CNN model trained with the full COCO dataset.
```

(continues on next page)

(continued from previous page)

```
Model used: fasterrcnn
Dataset information
    Name: COCO
    Number of classes: 80

Creation date: 2018-04-17T16:58:00.658815
Luminoth version: v0.1.1

Source: remote (DOWNLOADED)
URL: https://github.com/tryolabs/luminoth/releases/download/v0.1.0/e1c2565b51e9.tar
```

You can see that this dataset consists of 80 classes, and other useful information. Let's see what the `fast` checkpoint is about: `lumi checkpoint info aad6912e94d9` or `lumi checkpoint info fast`

```
SSD w/Pascal VOC (aad6912e94d9, fast)
Base SSD model trained with the full Pascal dataset.

Model used: ssd
Dataset information
    Name: Pascal VOC
    Number of classes: 20

Creation date: 2018-04-12T17:42:01.598779
Luminoth version: v0.1.1

Source: remote (NOT_DOWNLOADED)
URL: https://github.com/tryolabs/luminoth/releases/download/v0.1.0/aad6912e94d9.tar
```

If you want to get predictions for an image or video using a specific checkpoint (for example, `fast`) you can do so by using the `--checkpoint` parameter:

```
lumi predict bicycling-1160860_1280.jpg --checkpoint fast -f preds/objects.json -d␣
↪preds/
```

Inspecting the image, you'll see that it doesn't work as nicely as the `accurate` checkpoint.

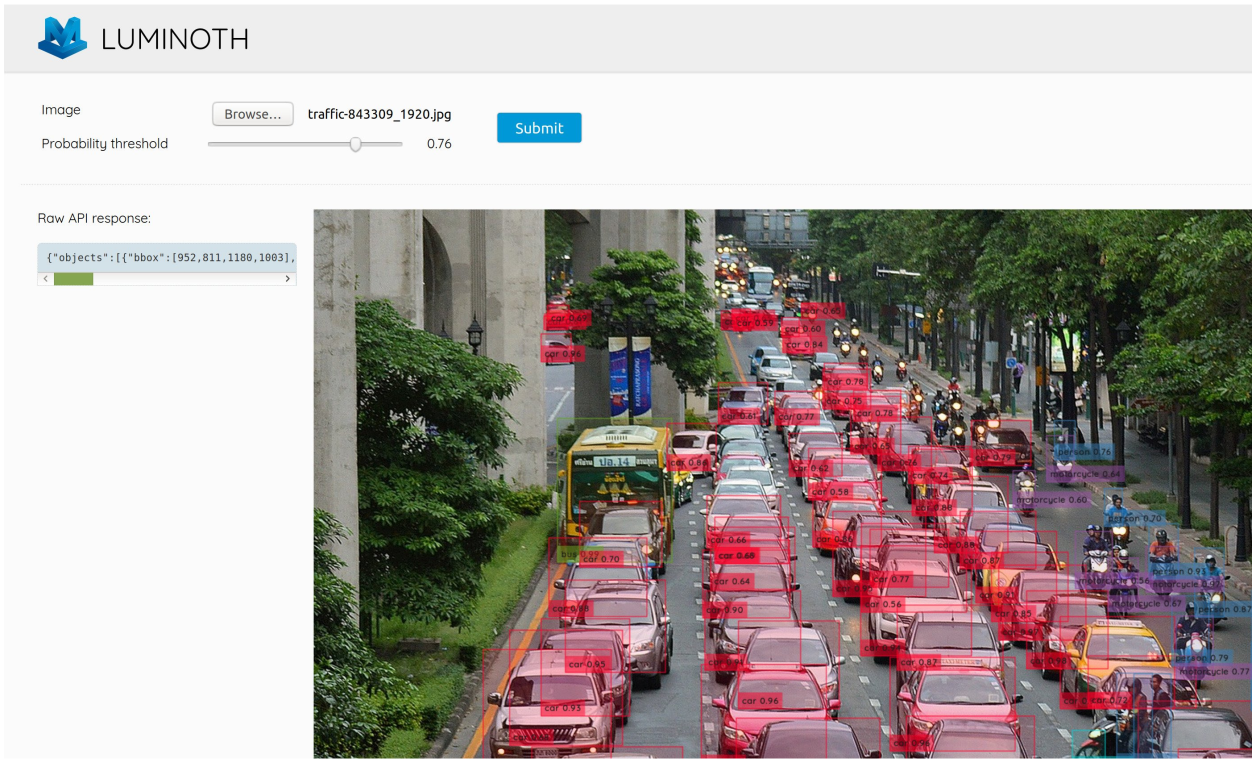
Also note that in every command where we used the alias of checkpoint, we could also have used the id.

### The built-in interface for playing around

Luminoth also includes a simple web frontend so you can play around with detected objects in images using different thresholds.

To launch this, simply type `lumi server web` and then open your browser at http://localhost:5000. If you are running on an external VM, you can do `lumi server web --host 0.0.0.0 --port <port>` to open in a custom port.

Now, select an image and submit! See the results.



```
{"objects":[{"bbox":[952,811,1180,1003],
```

You can go ahead and change the "probability threshold" slidebar and see how the detection looks with more or less filtering. You'll see that as you lower the threshold, more objects appear (and many times these are incorrect), while increasing the threshold makes the most accurate guesses but misses many of the objects you wish to detect.

---

Next: *Building custom traffic dataset*

### 1.3.2 Building custom traffic dataset

Even though pre-trained checkpoints are really useful, most of the time you will want to train an object detector using your own dataset. For this, you need a source of images and their corresponding bounding box coordinates and labels, in some format that Luminoth can understand. In this case, we are interested in street traffic related objects, so we will need to source images relevant to our niche.

#### How Luminoth handles datasets

Luminoth reads datasets natively only in TensorFlow's TFRecords format. This is a binary format that will let Luminoth consume the data very efficiently.

In order to use a custom dataset, you must first transform whatever format your data is in, to TFRecords files (one for each split — train, val, test). Fortunately, Luminoth provides several CLI tools (see *Adapting a dataset*) for transforming popular dataset format (such as Pascal VOC, ImageNet, COCO, CSV, etc.) into TFRecords. In what follows, we will leverage this.

#### Building a traffic dataset using OpenImages

OpenImages V4 is the largest existing dataset with object location annotations. It contains 15.4M bounding-boxes for 600 categories on 1.9M images, making it a very good choice for getting example images of a variety of (not niche-domain) classes (persons, cars, dolphin, blender, etc).

### Preparing the data

We should start by downloading the annotation files (this and this, for train) and the class description file. Note that the files with the annotations themselves are pretty large, totalling over 1.5 GB (and this CSV files only, without downloading a single image!).

After we get the `class-descriptions-boxable.csv` file, we can go over all the classes available in the OpenImages dataset and see which ones are related to **traffic**. The following were hand-picked after examining the full file:

```
/m/015qff,Traffic light
/m/0199g,Bicycle
/m/01bjv,Bus
/m/01g317,Person
/m/04_sv,Motorcycle
/m/07r04,Truck
/m/0h2r6,Van
/m/0k4j,Car
```

### Using the Luminoth dataset reader

Luminoth includes a **dataset reader** that can take OpenImages format. As the dataset is so large, this will never download every single image, but fetch only those we want to use and store them directly in the TFRecords file.

Note that the dataset reader expects a particular directory layout so it knows where the files are located. In this case, files corresponding to the examples must be in a folder named like their split (*train*, *test*, . . . ). So, you should have the following:

```
.
├── class-descriptions-boxable.csv
└── train
    ├── train-annotations-bbox.csv
    └── train-annotations-human-imagelabels-boxable.csv
```

Next, run the following command:

```
lumi dataset transform \
      --type openimages \
      --data-dir . \
      --output-dir ./out \
      --split train  \
      --class-examples 100 \
      --only-classes=/m/015qff,/m/0199g,/m/01bjv,/m/01g317,/m/04_sv,/m/07r04,/m/0h2r6,
↪/m/0k4j
```

This will generate TFRecord file for the `train` split. You should get something like this in your terminal after the command finishes:

```
INFO:tensorflow:Saved 360 records to "./out/train.tfrecords"
INFO:tensorflow:Composition per class (train):
INFO:tensorflow:        Person (/m/01g317): 380
INFO:tensorflow:        Car (/m/0k4j): 255
INFO:tensorflow:        Bicycle (/m/0199g): 126
```

(continues on next page)

(continued from previous page)

```
INFO:tensorflow:        Bus (/m/01bjv): 106
INFO:tensorflow:        Traffic light (/m/015qff): 105
INFO:tensorflow:        Truck (/m/07r04): 101
INFO:tensorflow:        Van (/m/0h2r6): 100
INFO:tensorflow:        Motorcycle (/m/04_sv): 100
```

Apart from the TFRecord file, you will also get a `classes.json` file that lists the names of the classes in your dataset.

Note that:

- As we are using `--only-classes`, so we filter to only use the classes we care about.
- We are using `--max-per-class` of 100. This setting will make it stop when every class has at least 100 examples. However, some classes may end up with many more; for example here it needed to get 380 instances of persons to get 100 motorcycles, considering the first 360 images.
- We could also have used `--limit-examples` option so we know the number of records in our final dataset beforehand.

Of course, this dataset is **way too small** for any meaningful training to go on, but we are just showcasing. In real life, you would use a much larger value for `--max-per-class` (ie. 15000) or `--limit-examples`.

---

Next: *Training the model*

## 1.3.3 Training the model

Now that we have created our (toy) dataset, we can proceed to train our model.

### The configuration file

Training orchestration, including the model to be used, the dataset location and training schedule, is specified in a YAML config file. This file will be consumed by Luminoth and merged to the default configuration, to start the training session.

You can see a minimal config file example in sample_config.yml. This file illustrates the entries you'll most probably need to modify, which are:

- `train.run_name`: the run name for the training session, used to identify it.
- `train.job_dir`: directory in which both model checkpoints and summaries (for TensorBoard consumption) will be saved. The actual files will be stored under `<job_dir>/<run_name>`.
- `dataset.dir`: directory from which to read the TFRecord files.
- `model.type`: model to use for object detection (`fasterrcnn`, or `ssd`).
- `network.num_classes`: number of classes to predict (depends on your dataset).

For looking at all the possible configuration options, mostly related to the model itself, you can check the base_config.yml file.

### Building the config file for your dataset

Probably the most important setting for training is the **learning rate**. You will most likely want to tune this depending on your dataset, and you can do it via the `train.learning_rate` setting in the configuration. For example, this would be a good setting for training on the full COCO dataset:

```
learning_rate:
  decay_method: piecewise_constant
  boundaries: [250000, 450000, 600000]
  values: [0.0003, 0.0001, 0.00003, 0.00001]
```

To get to this, you will need to run some experiments and see what works best.

```
train:
  # Run name for the training session.
  run_name: traffic
  job_dir: <change this directory>
  learning_rate:
    decay_method: piecewise_constant
    # Custom dataset for Luminoth Tutorial
    boundaries: [90000, 160000, 250000]
    values: [0.0003, 0.0001, 0.00003, 0.00001]
dataset:
  type: object_detection
  dir: <directory with your dataset>
model:
  type: fasterrcnn
  network:
    num_classes: 8
  anchors:
    # Add one more scale to be better at detecting small objects
    scales: [0.125, 0.25, 0.5, 1, 2]
```

### Running the training

Assuming you already have both your dataset (TFRecords) and the config file ready, you can start your training session by running the command as follows:

```
lumi train -c config.yml
```

You can use the `-o` option to override any configuration option using dot notation (e.g. `-o model.rpn.proposals.nms_threshold=0.8`).

If you are using a CUDA-based GPU, you can select the GPU to use by setting the `CUDA_VISIBLE_DEVICES` environment variable (see here for more info).

When the training is running, you should see Luminoth print out for each step, the minibatch (single image), and the training loss related to that minibatch.

Image to image, the training loss will jump around, and this is expected. However, the trend will be that the loss will gradually start to decrease. For this, it is interesting to look at it using tools like TensorBoard.

### Storing partial weights (checkpoints)

As the training progresses, Luminoth will periodically save a checkpoint with the current weights of the model. These weights let you resume training from where you left off!

The files will be output in your `<job_dir>/<run_name>` folder. By default, they will be saved every 600 seconds of training, but you can configure this with the `train.save_checkpoint_secs` setting in your config file.

The default is to only store the latest checkpoint (that is, when a checkpoint is generated, the previous checkpoint gets deleted) in order to conserve storage. You might find the `train.checkpoints_max_keep` option in your train YML configuration useful if you want to keep more checkpoints around.

---

Next: *Using TensorBoard to visualize the training process*

### 1.3.4 Using TensorBoard to visualize the training process

Now that the training is running, you should pay special attention to how it is progressing, to make sure that your model is actually learning something.

TensorBoard is a very good tool for this, allowing you to see plenty of plots with the training related metrics. By default, Luminoth writes TensorBoard summaries during training, so you can leverage this tool without any effort!

To run it, you can use:

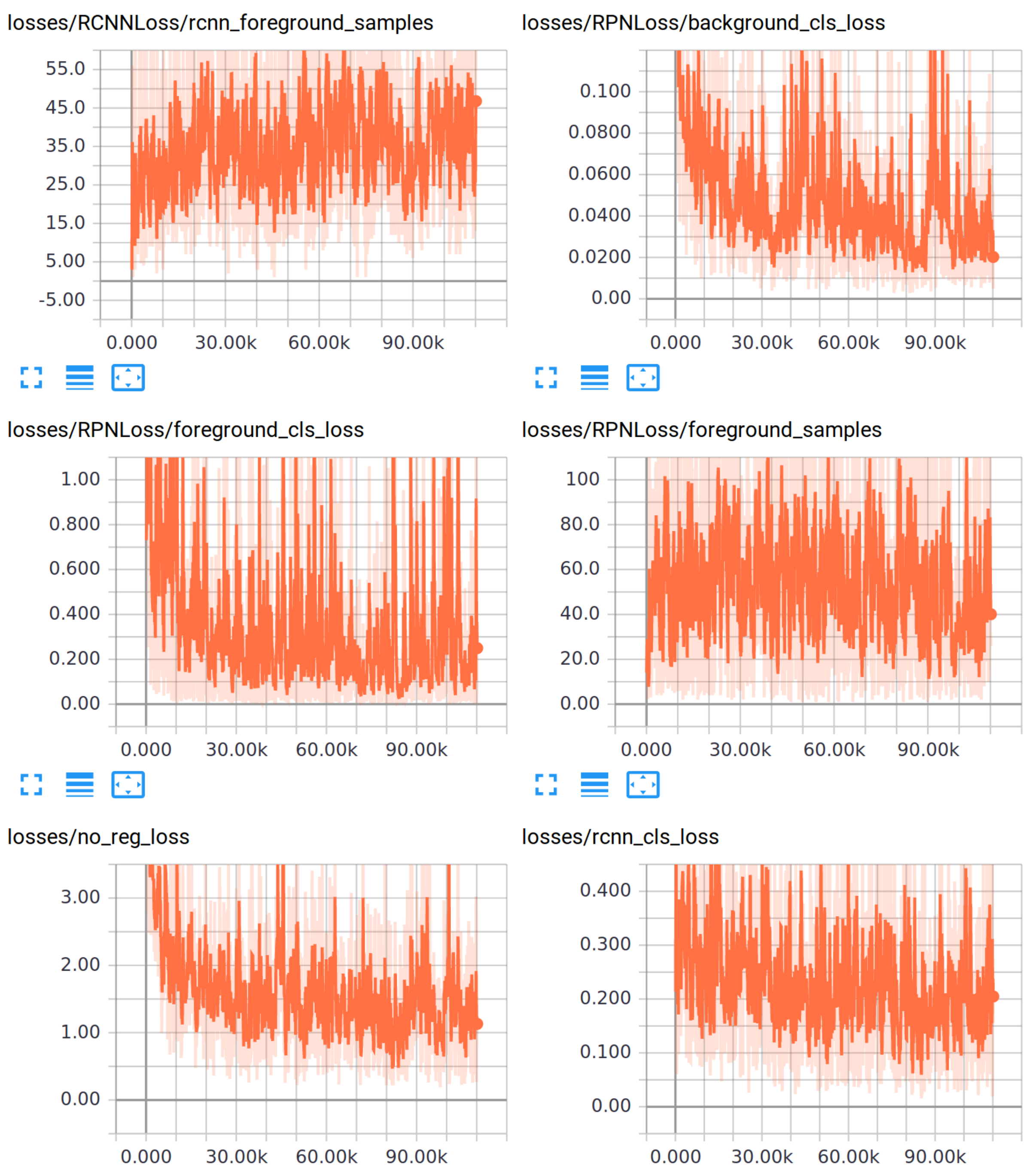
```
tensorboard --logdir <job_dir>/<run_name>
```

If you are running from an external VM, make sure to use `--host 0.0.0.0` and `--port` if you need other one than the default 6006.

#### What to look for

First, go to the "Scalars" tab. You are going to see several *tags*, but in this case, you only should care about some of the metrics behind `losses`.

The loss is your objective function, which you want to minimize. In the case of Faster R-CNN, we have a model with a multi-objective loss, ie. the model is trying to minimize several things at the same time. This is why you will see several plots here.

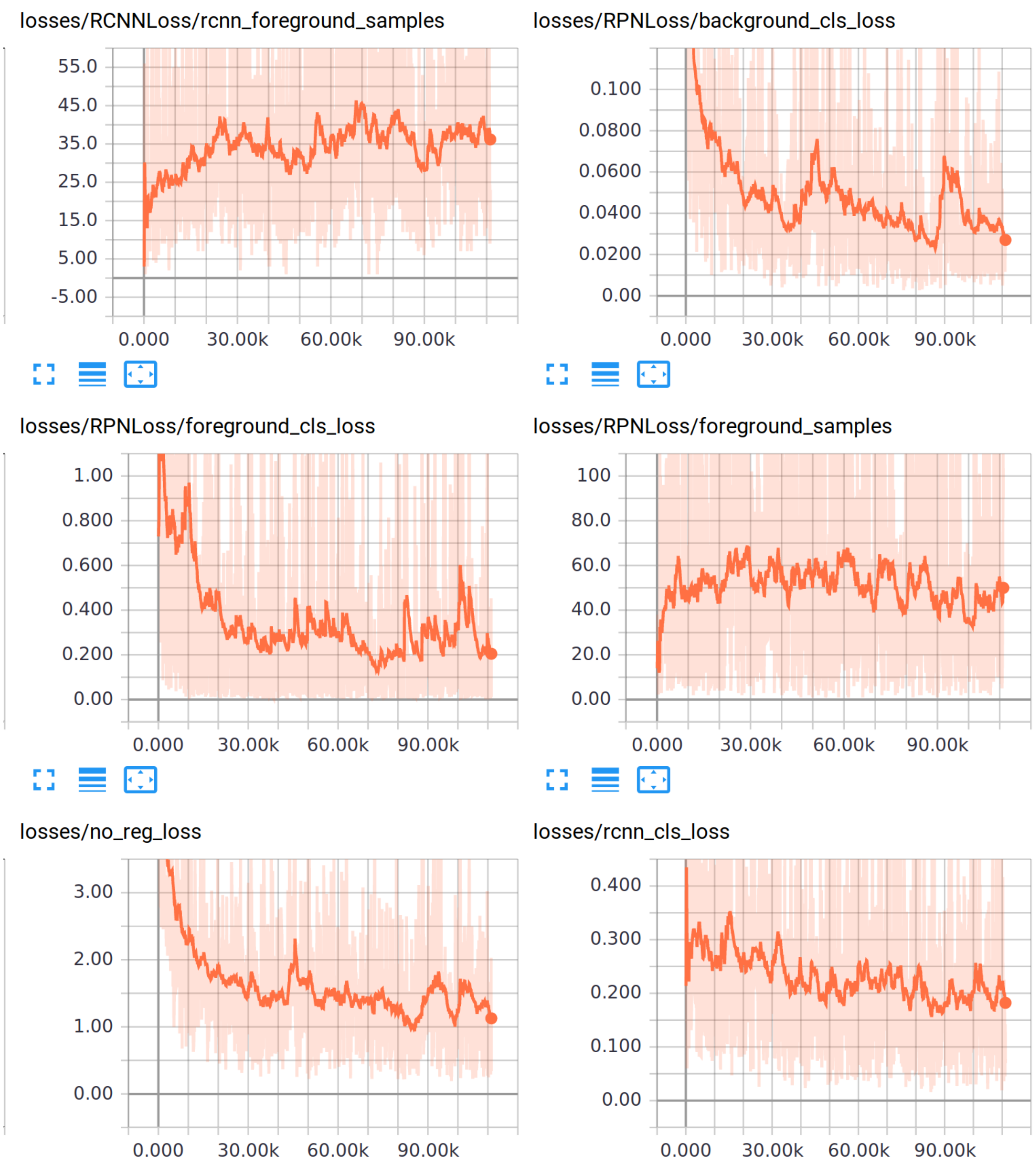


You should mostly be interested in the one called `no_reg_loss`. This is the total loss function, without including the regularization loss (which will always decrease). Therefore, it will give you a nice summary of how the training is progressing.

Your job is to make sure this `no_reg_loss` value is going decreasing during training.

As we said before, the loss will jump around a lot, because each point corresponds to a minibatch, which in this case is a single image. A good prediction in a particular image will yield a low loss, however, if the model performed poorly another particular image, the loss will be very high.

To help you notice the trend, you can set **Smoothing** to a higher value. For example, setting it so 0.95 the plots now look like this:



Now it's much more clear: at first, there is a sharp drop in the loss, but then it is not as noticeable.

## Tips: how to do training and tune hyperparameters

To get the best result for your dataset, you are going to have to run several training runs until you nail it. Here is what we have done in the past:

1. Start with a **fixed learning rate**. The "magical" value of 0.0003 has worked for us for a variety of problems.
2. Run the training until the loss sort of **stabilizes** for some time (many thousands of steps).

3. After the loss is roughly stable, **decrease** the learning rate. For example, you might choose a different value like 0.001. In the case of Faster R-CNN model, we (approximately) divide the learning rate by 3 (0.0003, 0.0001, 0.00003, . . . ).
4. You should see the loss leave this plateau and become even smaller. If so, good job! Notice the approximate step number in which you would consider that the loss stabilized.

This information will let you build a training configuration that is good for your dataset. For example, you can better tune your configuration for learning rate:

```
train:
  learning_rate:
    decay_method: piecewise_constant
    boundaries: [steps_1, steps_2, ..., steps_n]
    values: [value_0, value_1, value_2, ..., value_n]
```

### Manually inspecting how model performs with lumi server web

You can also use `lumi server web` command that we have seen before and try your partially trained model in a bunch of novel images.

For this, you can launch it with a config file like:

```
lumi server web -c config.yml
```

Remember that here you can also use `--host` and `--port` options should you happen to need those.

---

Next: *Evaluating models*

## 1.3.5 Evaluating models

As you are training the model, your job is to make the training loss decrease.

However, it might be the case that your model is learning to fit your training data very well, but it won't work as well when it is fed new, unseen data. This is called *overfitting*, and to avoid it, it is very important to evaluate models in data they haven't seen during training.

Usually, datasets (like OpenImages, which we just used) provide "splits". The "train" split is the largest, and the one from which the model actually does the learning. Then, you have the "validation" (or "val") split, which consists of different images, in which you can draw metrics of your model's performance, in order to better tune your hyperparameters. Finally, a "test" split is provided in order to conduct the final evaluation of how your model would perform in the real world once it is trained.

### Building a validation dataset

Let's start by building TFRecords from the validation split of OpenImages. For this, we can download the files with the annotations and use the same `lumi dataset transform` that we used to build our training data.

In your dataset folder (where the `class-descriptions-boxable.csv` is located), run the following commands:

```
mkdir validation
wget -P validation https://storage.googleapis.com/openimages/2018_04/validation/
↪validation-annotations-bbox.csv
wget -P validation https://storage.googleapis.com/openimages/2018_04/validation/
↪validation-annotations-human-imagelabels-boxable.csv
```

After the downloads finish, we can build the TFRecords with the following:

```
lumi dataset transform \
      --type openimages \
      --data-dir . \
      --output-dir ./out \
      --split validation  \
      --class-examples 100 \
      --only-classes=/m/015qff,/m/0199g,/m/01bjv,/m/01g317,/m/04_sv,/m/07r04,/m/0h2r6,
↪/m/0k4j
```

Note that again, we are building a very reduced toy evaluation dataset by using `--class-examples` (as we did for training).

### The `lumi eval` command

In Luminoth, `lumi eval` will make a run through your chosen dataset split (ie. `validation` or `test`), and run the model through every image, and then compute metrics like loss and mAP.

This command works equivalenty to `lumi train`, so it will occupy your GPU and output summaries for TensorBoard.

If you are lucky and happen to have more than one GPU in your machine, it is advisable to run both `train` and `eval` at the same time. In this case, you can get things like your validation metrics in TensorBoard and watch them as you train.

Start by running the evaluation:

```
lumi eval --split validation -c custom.yml
```

Luminoth should now load the last available checkpoint, and start processing images. After it's done with a full pass through the split, it will output something like this in the shell:

```
...
386 processed in 244.44s (global 1.58 images/s, period 1.87 images/s)
426 processed in 265.03s (global 1.61 images/s, period 1.94 images/s)
465 processed in 285.33s (global 1.63 images/s, period 1.92 images/s)
INFO:tensorflow:Finished evaluation at step 271435.
INFO:tensorflow:Evaluated 476 images.
INFO:tensorflow:Average Precision (AP) @ [0.50] = 0.720
INFO:tensorflow:Average Precision (AP) @ [0.75] = 0.576
INFO:tensorflow:Average Precision (AP) @ [0.50:0.95] = 0.512
INFO:tensorflow:Average Recall (AR) @ [0.50:0.95] = 0.688
INFO:tensorflow:Evaluated in 294.92s
INFO:tensorflow:All checkpoints evaluated; sleeping for a moment
INFO:tensorflow:Found 0 checkpoints in run_dir with global_step > 271435
```

After the full pass, `eval` will sleep until a new checkpoint is stored in the same directory.

### The mAP metrics

Mean Average Precision (mAP) is the metric commonly used to evaluate object detection task. It computes how well your classifier works **across all classes**.

We are not going to go over how it works, but you can read this blog post for a nice explanation.

What you need to know is that mAP will be a number between 0 and 1, and the higher the better. Moreover, it can be calculated across different IoU (Intersection over Union) thresholds.

For example, Pascal VOC challenge metric uses 0.5 as threshold (notation mAP@0.5), and COCO dataset uses mAP at different thresholds and averages them all out (notation mAP@[0.5:0.95]). Luminoth will print out several of these metrics, specifying the thresholds that were used under this notation.

### Visualizing evaluation metrics in TensorBoard

If you fire up TensorBoard, you will see that you get new "tags" that come from the evaluation: in this case, we will get `validation_metrics` and `validation_losses`.

#### `validation_losses`

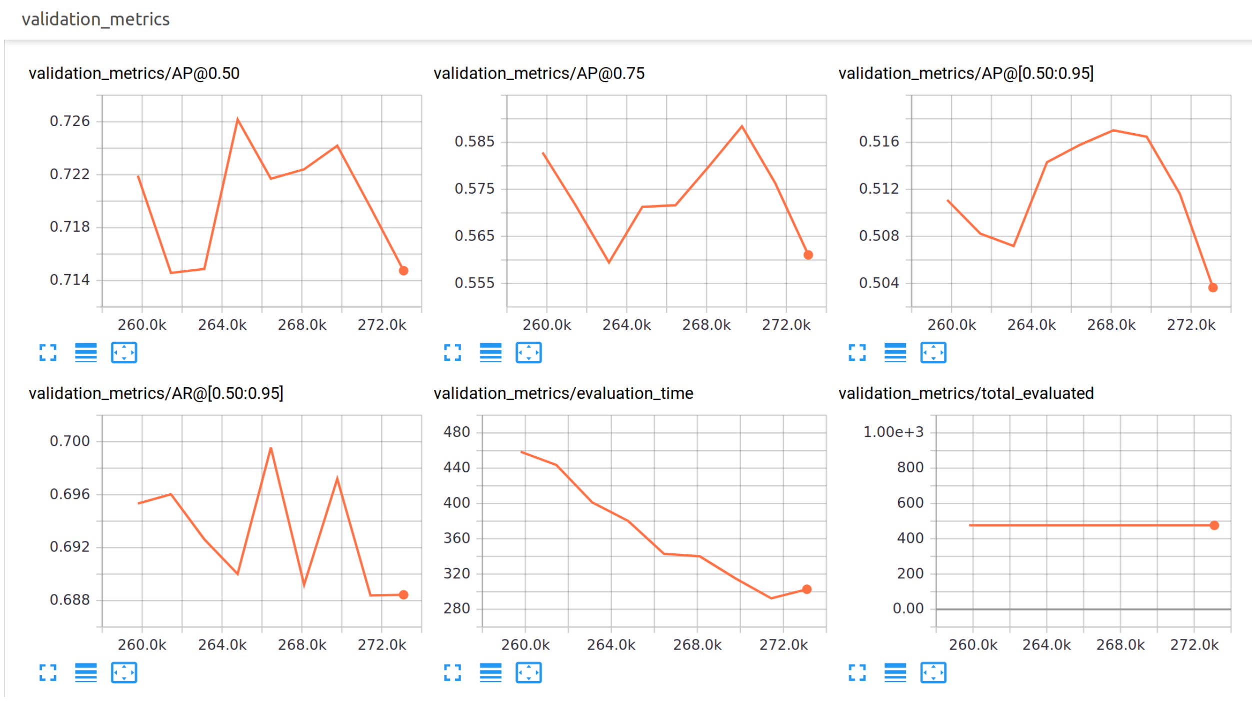
Here, you will get the same loss values that Luminoth computes for the train, but for the chosen dataset split (validation, in this case).

As in the case of train, you should mostly look at `validation_losses/no_reg_loss`. As long as it goes down, you know the model is learning.

If the training loss keeps decreasing but validation loss does not, you know that your model is no longer learning anything useful and can thus stop the training. If your validation loss starts to increase, then you know your model is overfitting.

#### `validation_metrics`

These will be the mAP metrics that will help you judge how well your model perform.

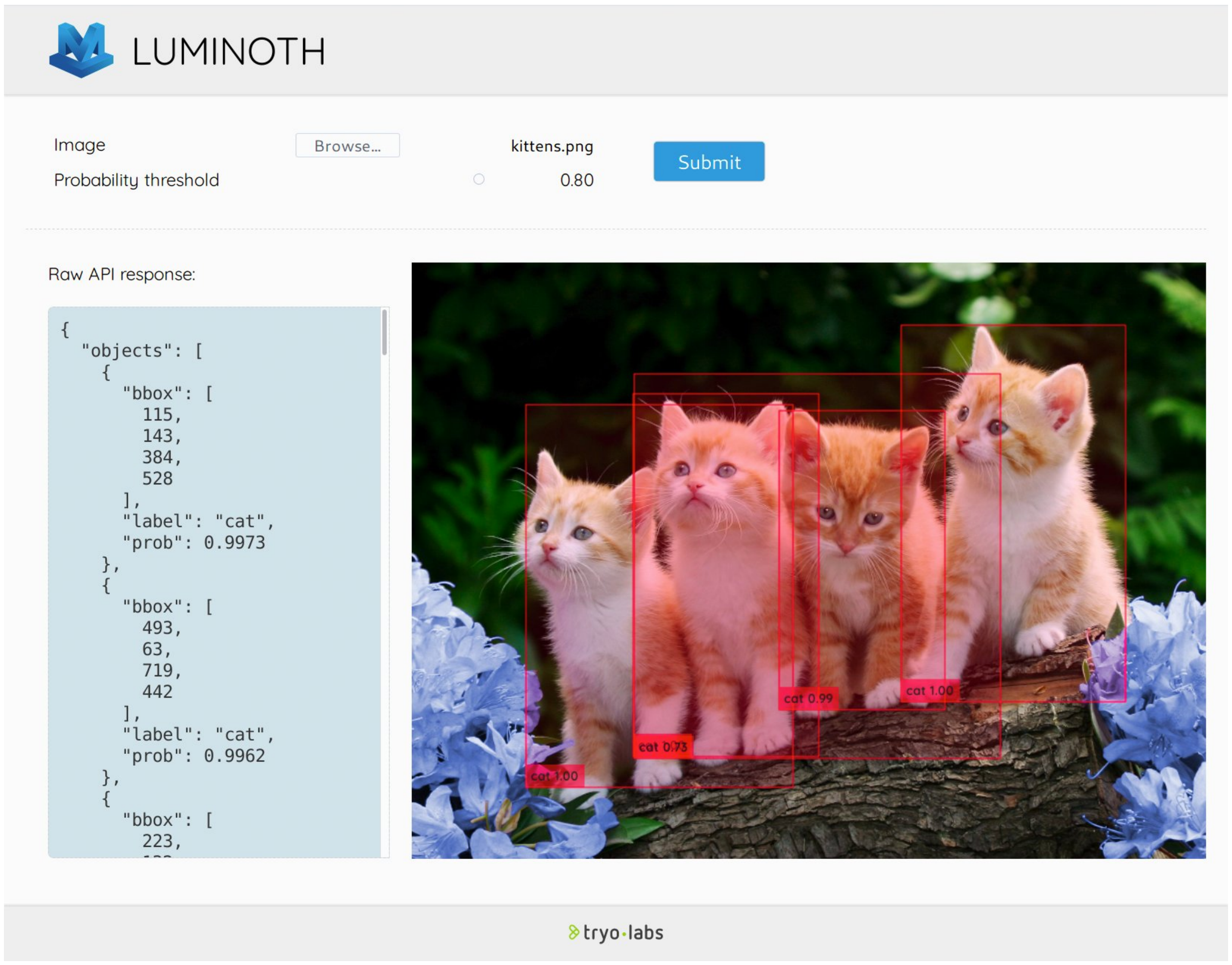


For viewing these plots, some important considerations:

- Unlike with the other metrics, you do not want to use Smoothing here. The mAP values refer to the entire dataset split, so it will not jump around as much as other metrics.
- Click "Fit domain to data" (third blue button in the bottom left of each plot) in order to see the full plot.

## Visual inspection of the model

As another reminder, do not forget that it is crucially important that you verify that the model is working properly by doing manual inspection of the results. You will find `lumi server web` very useful for this.



```
{
  "objects": [
    {
      "bbox": [
        115,
        143,
        384,
        528
      ],
      "label": "cat",
      "prob": 0.9973
    },
    {
      "bbox": [
        493,
        63,
        719,
        442
      ],
      "label": "cat",
      "prob": 0.9962
    },
    {
      "bbox": [
        223,
```

mAP numbers are good as a summary, but inspecting the model's behavior will let you discover specific cases that are not working and could be improved by tuning some other hyperparameter. For example, you might add more anchor scales if the sizes of your objects varies a lot.

---

Next: *Creating and sharing your own checkpoints*

### 1.3.6 Creating and sharing your own checkpoints

After the model is trained to your satisfaction, it is very useful to actually create a **checkpoint** that makes it straightforward to use your model.

#### Creating a checkpoint

We can create checkpoints and set some metadata like name, alias, etc. This time, we are going to create the checkpoint for our traffic model:

```
lumi checkpoint create \
    config.yml \
    -e name="OpenImages Traffic" \
    -e alias=traffic
```

After running this, you should get an output similar to this:

```
Creating checkpoint for given configuration...
Checkpoint cb0e5d92a854 created successfully.
```

You can verify that you do indeed have the checkpoint when running `lumi checkpoint list`, which should get you an output similar to this:

```
================================================================================
|           id |                  name |       alias | source |         status |
================================================================================
| e1c2565b51e9 |   Faster R-CNN w/COCO |    accurate | remote |     DOWNLOADED |
| aad6912e94d9 |      SSD w/Pascal VOC |        fast | remote |     DOWNLOADED |
| cb0e5d92a854 |     OpenImages Traffic |     traffic |  local |          LOCAL |
================================================================================
```

Moreover, if you inspect the `~/.luminoth/checkpoints/` folder, you will see that now you have a folder that corresponds to your newly created checkpoint. Inside this folder are the actual weights of the model, plus some metadata and the configuration file that was used during training.

#### Sharing checkpoints

#### Exporting a checkpoint as a single file

Simply run `lumi checkpoint export cb0e5d92a854`. You will get a file named `cb0e5d92a854.tar` in your current directory, which you can easily share to somebody else.

#### Importing a checkpoint file

By running `lumi checkpoint import cb0e5d92a854.tar`, the checkpoint will be listed locally. Note that this will fail if the checkpoint already exists, as expected (you can use `lumi checkpoint delete` if you want to try this anyway).

You can now use it very easily, for example we can reference our checkpoint using its alias by running `lumi server web --checkpoint traffic`. Neat!

---

Next: *Using Luminoth from Python*

### 1.3.7 Using Luminoth from Python

Calling Luminoth from your Python app is very straightforward. You can even make use of helper functions to visualize the bounding boxes.

```python
from luminoth import Detector, read_image, vis_objects

image = read_image('traffic-image.png')

# If no checkpoint specified, will assume `accurate` by default. In this case,
# we want to use our traffic checkpoint. The Detector can also take a config
# object.
detector = Detector(checkpoint='traffic')

```

(continues on next page)

(continued from previous page)

```
# Returns a dictionary with the detections.
objects = detector.predict(image)

print(objects)

vis_objects(image, objects).save('traffic-out.png')
```

---

This was the end of the tutorial! Hope you enjoyed :)

## 1.4 Adapting a dataset

If a pre-trained checkpoint for the task you want to perform is not available, you can train Luminoth with an existing open dataset, or even your own.

The first step before training a model with Luminoth is to convert your dataset to TensorFlow's `.tfrecords` format. This ensures that no matter what image or annotation formats the original dataset uses, it will be transformed into something that Luminoth can understand and process efficiently, either while training locally or in the cloud.

For this purpose, Luminoth provides a conversion tool which includes support for some of the most well-known datasets for the object detection task.

### 1.4.1 Conversion tool

As explained above, the conversion tool, invoked with the command `lumi dataset transform`, allows you to transform a dataset in a standard format into one that can be understood by Luminoth.

#### Supported datasets

You can select the annotation scheme used by your dataset with the `--type` option. As long as your dataset follows the same scheme, the conversion tool will be able to transform it correctly. Furthermore, you can write your own conversion tool to read a custom format (see *Supporting your own dataset format*).

The supported types are:

- `pascal`: format used by the Pascal VOC dataset.
- `imagenet`: format used by the ImageNet dataset.
- `coco`: format used by the COCO dataset.
- `openimages`: format used by the OpenImages dataset.
- `csv`: specify the bounding boxes using a CSV file with one annotation per line.

#### Input and output

In order to point the conversion tool to the actual data to transform, you must set the `--data-dir` option to the directory containing it. This path should follow the directory structure expected by the indicated `--type`. For instance, in the case of the `pascal` dataset type, this will be the `VOCdevkit/VOC2007` directory obtained from extracting the tar file provided by the dataset page.

The output directory is specified with the `--output-dir` option. Inside it, one TFrecords file per dataset split will be stored. This file may be very, very large, depending on the dataset, so make sure there's enough space in the disk.

You can also specify which dataset splits (i.e. train, validation or test) to convert, whenever that information is available. You can do so by using the `--split <train|val|test>` option, using it more than once if you want to transform more than one split at the same time.

#### Limiting the dataset

For datasets with many classes, you might want to ignore some of them when training a custom detector. For instance, if you want to train a traffic detector, you could start with the COCO dataset but only use, out of the eighty classes present in it, cars, trucks, buses and motorcycles. You can do so with the `--only-classes` option, by passing a comma-separated list of classes to keep in the final dataset.

Moreover, if you wish to use several classes but not the entire set of images available in a (possibly large) dataset, you may use the `--class-examples` option.

During development, it is often useful to verify that the model can actually overfit a small dataset. You can create such a dataset by using the `--limit-examples` option.

#### Examples

Say we want to transform the `train` and `val` splits of the Pascal VOC2012 dataset. This will output the corresponding `.tfrecords` files to the output dir:

```
$ lumi dataset transform \
        --type pascal \
        --data-dir datasets/pascal/VOCdevkit/VOC2012/ \
        --output-dir datasets/pascal/tf/ \
        --split train --split val
```

If we wanted to use COCO to create a dataset with vehicles and people (say, for our up-and-coming self-driving car), we could use the following command:

```
$ lumi dataset transform \
        --type coco \
        --data-dir datasets/coco/ \
        --output-dir datasets/coco/tf/ \
        --split train --split val
        --only-classes=car,truck,bus,motorcycle,bicycle,person
```

#### Supporting your own dataset format

TODO: Guidelines on how to write your own dataset reader.

For now, you can see `luminoth/tools/dataset/readers/object_detection/pascalvoc.py` as an example on creating your own reader.

### 1.4.2 Merge tool

Sometimes you don't have a dataset for your model, but are able to leverage data from several open datasets. Luminoth provides a dataset merging tool for this purpose, allowing you to combine several TFrecords files (i.e. already converted into Luminoth's expected format) into a single one.

This tool is provided through the `lumi dataset merge` command, which receives a list of TFrecords files and outputs it to the file indicated by the last argument. For example:

```
$ lumi dataset merge \
        datasets/pascal/tf/2007/only-traffic/train.tfrecords \
        datasets/pascal/tf/2012/only-traffic/train.tfrecords \
        datasets/coco/tf/only-traffic/train.tfrecords \
        datasets/tf/train.tfrecords
```

## 1.5 Training your own model

In order to train your own model, two things are required:

- A dataset ready to be consumed by Luminoth (see *Adapting a dataset*).
- A configuration file for the run.

We'll start by covering the configuration file, then proceed to the training itself, both locally and in the cloud.

### 1.5.1 Configuration

Training orchestration, including the model to be used, the dataset location and training schedule, is specified in a YAML config file. This file will be consumed by Luminoth and merged to the default configuration to start the training session.

You can see a minimal config file example in sample_config.yml. This file illustrates the entries you'll most probably need to modify, which are:

- `train.run_name`: The run name for the training session, used to identify it.
- `train.job_dir`: Directory in which both model checkpoints and summaries (for Tensorboard consumption) will be saved. The actual files will be stored under `{job_dir}/{run_name}`, so serving `{job_dir}` with Tensorboard will allow you to see all your runs at once.
- `dataset.dir`: Directory from which to read the TFrecords files.
- `model.type`: Model to use for object detection (e.g. `fasterrcnn`, `ssd`).
- `network.num_classes`: Number of classes to predict.

There are a great deal of configuration options, mostly related to the model itself. You can, for instance, see the full range of options for the Faster R-CNN model, along with a brief description of each, in its base_config.yml file.

### 1.5.2 Training

The model training itself can either be run locally (on the CPU or GPU available) or in Google Cloud's Cloud ML Engine.

#### Locally

Assuming you already have both your dataset and the config file ready, you can start your training session by running the command as follows:

```
$ lumi train -c my_config.yml
```

The `lumi train` CLI tool provides the following options related to training.

- `--config`/`-c`: Config file to use. If the flag is repeated, all config files will be merged in left-to-right order so that every file overwrites the configuration of keys defined previously.
- `--override`/`-o`: Override any configuration setting using dot notation (e.g.: `-o model.rpn.proposals.nms_threshold=0.8`).

If you're using a CUDA-based GPU, you can select the GPU to use by setting the `CUDA_VISIBLE_DEVICES` environment variable. (See the NVIDIA site for more information.)

You can run Tensorboard on the `job_dir` to visualize training, including the loss, evaluation metrics, training speed, and even partial images.

#### Google Cloud

Luminoth can easily run in Google Cloud ML Engine with a single command.

For more information, see usage/cloud.

## 1.6 Evaluating a model

## 1.7 Cloud management

We support training in Google Cloud ML engine, which has native Tensorflow support.

Instead of building Python packages yourself and using Google Cloud SDK, we baked the process inside Luminoth itself, so you can pull it of with a few simple commands.

You can choose how many workers you want, which scale tiers to use, and where to store the results. We also provide some utilities to monitor and manage your job right from your command line.

For all the cloud functionalities, the files read (such as the datasets in TFRecord format) and written (such as logs and checkpoints) by Luminoth will reside in Buckets instead of your local disk.

### 1.7.1 Pre-requisites

1. Create a Google Cloud project.
2. Install Google Cloud SDK on your machine. Although it is not strictly necessary, it will be useful to enable the required APIs and upload your dataset.
3. Login with Google Cloud SDK:

```
gcloud auth login
```

4. **Enable the following APIs:**
    - Compute Engine
    - Cloud Machine Learning Engine
    - Google Cloud Storage

   You can do it through the web console or with the following command:

```
gcloud services enable compute.googleapis.com ml.googleapis.com storage-component.
↪googleapis.com
```

Be patient, it can take a few minutes!

5. Upload your dataset's TFRecord files to a Cloud Storage bucket:

```
gsutil -o GSUtil:parallel_composite_upload_threshold=150M cp -r /path/to/dataset/
↪tfrecords gs://your_bucket/path
```

6. Create a Service Account Key (JSON format) and download it to your directory of choice. You may add it as an Editor of your project. If necessary, add required roles (permissions) to your service account.
7. Point the `GOOGLE_APPLICATION_CREDENTIALS` environment variable to the JSON file of the service account.

### 1.7.2 Running a training job

Luminoth command line tool provides commands to submit training jobs, list them and fetch their logs.

`lumi cloud gc train` - Submit a training job.

**Required arguments:**

- `--config`: YAML configuration file for use in training.

**Optional arguments:**

- `--dataset`: full path to bucket with the dataset's TFRecord files, ie. `gs://<bucket_name>/<path>`. If not present, will default from the value specified in the YAML config file (`dataset.dir`).
- `--resume`: Id of the previous job to resume (start from last stored checkpoint). In case you are resuming multiple times, must always point to the first job (ie. the one that first created the checkpoint). - `--bucket`: Bucket name for storing data for the job, such as the logs. Defaults to `luminoth-<client_id>`.
- `--job-id`: Identifies the training job in Google Cloud. Defaults to `train_<timestamp>`.
- `--region`: Google Cloud region in which to set up the cluster.
- `--scale-tier`: Cluster configuration. Default: `BASIC_GPU`.
- `--master-type`: Master node machine type.
- `--worker-type`: Worker node machine type.
- `--worker-count`: Number of workers.
- `--parameter-server-type`: Parameter server node machine type.
- `--parameter-server-count`: Number of parameter servers.

Example:

```
lumi cloud gc train \
    --bucket luminoth-train-jobs \
    --dataset gs://luminoth-train-datasets/coco/tfrecords \
    -c config.yml
```

### 1.7.3 Resuming a previous training job

Sometimes, you may wish to restart a previous training job without losing all the progress made so far (ie. resume from checkpoint). For example, it might be the case that you have updated your TFRecords dataset and want your model fine-tuned with the new data.

The way to achieve this in Google Cloud is by launching a **new training job**, but telling Luminoth to resume a previous job id:

```
lumi cloud gc train \
    --resume <previous-job-id> \
    --bucket luminoth-train-jobs \
    --dataset gs://luminoth-train-datasets/coco/tfrecords \
    -c config.yml
```

**Keep in mind that for this to work:**

- `bucket` must match the same bucket name that was used for the job you are resuming.
- In case you are resuming a job multiple times, `previous-job-id` must be the id of the job that first created the checkpoint. This is so Luminoth keeps writing the new files to the same folder.

### 1.7.4 Listing jobs

`lumi cloud gc jobs` - List project's jobs.

**Optional arguments:**

- `--running`: Show running jobs only.

### 1.7.5 Fetching logs

`lumi cloud gc logs` - Fetch logs for a specific job.

**Required arguments:**

- `--job-id`: id of the training job, obtained after running `lumi cloud gc train`.

**Optional arguments:**

- `--polling-interval`: Seconds between each log request.

### 1.7.6 Running an evaluation job

`lumi cloud gc evaluate` - Submit an evaluation job.

**Required arguments:**

- `--train-folder`: Complete path (bucket included) where the training results are stored (config.yml should live here).

**Optional arguments:**

- `--split`: Dataset split to use. Defaults to `val`.
- `--job-id`: Job Id for naming the folder where the results of the evaluation will be stored.
- `--bucket`: The bucket where the evaluation results were stored.
- `--region`: Google Cloud region in which to run the job.

- `--scale-tier`: Cluster configuration. Default: `BASIC_GPU`.
- `--master-type`: Master node machine type.
- `--rebuild`: Whether to rebuild the package with the currently installed version of Luminoth, or use the same Luminoth package that was used for training.

Example:

```
lumi cloud gc evaluate \
    --train-folder gs://luminoth-train-jobs/lumi_train_XXXXXXXX_YYYYYY \
    --bucket luminoth-eval-jobs \
    --split test
```

### 1.7.7 Results

Everything related to a job is stored in its own folder on the bucket provided under the name `lumi_{job_id}`.

## 1.8 Working with checkpoints

TODO: Explain the rationale behind checkpoints, and expand each section.

List the checkpoints available on the system:

```
$ lumi checkpoint list
================================================================================
|           id |                  name |       alias | source |         status |
================================================================================
| 48ed2350f5b2 |   Faster R-CNN w/COCO |    accurate | remote | NOT_DOWNLOADED |
| e3256ffb7e29 |      SSD w/Pascal VOC |        fast | remote | NOT_DOWNLOADED |
================================================================================
```

Inspect a checkpoint:

```
$ lumi checkpoint info accurate
Faster R-CNN w/COCO (48ed2350f5b2, accurate)
Base Faster R-CNN model trained with the full COCO dataset.

Model used: fasterrcnn
Dataset information
    Name: COCO
    Number of classes: 80

Creation date: 2018-03-21T20:04:59.785711
Luminoth version: v0.1.0

Source: remote (NOT_DOWNLOADED)
URL: https://github.com/tryolabs/luminoth/releases/download/v0.0.3/48ed2350f5b2.tar
```

Refresh the remote checkpoint index:

```
$ lumi checkpoint refresh
Retrieving remote index... done.
2 new remote checkpoints added.
```

Download a remote checkpoint:

```
$ lumi checkpoint download accurate
Downloading checkpoint...  [####################################]  100%
Importing checkpoint... done.
Checkpoint imported successfully.
```

Create a checkpoint:

```
$ lumi checkpoint create config.yml -e name='Faster R-CNN with cars' -e alias=cars
Creating checkpoint for given configuration...
Checkpoint b5c140450f48 created successfully.
```

Edit a checkpoint:

```
$ lumi checkpoint edit b5c140450f48 -e 'description=Model trained with COCO cars.'
```

Delete a checkpoint:

```
$ lumi checkpoint delete b5c140450f48
Checkpoint b5c140450f48 deleted successfully.
```

Export a checkpoint into a tar file, for easy sharing:

```
$ lumi checkpoint export 48ed2350f5b2
Checkpoint 48ed2350f5b2 exported successfully.
```

Import a previously-exported checkpoint:

```
$ lumi checkpoint import 48ed2350f5b2.tar
```

## 1.9 Datasets

## 1.10 Models

**class** luminoth.models.base.**BaseNetwork**(*config*, *name='base_network'*)

Convolutional Neural Network used for image classification, whose architecture can be any of the *VALID_ARCHITECTURES*.

This class wraps the *tf.slim* implementations of these models, with some helpful additions.

**_build**(*inputs*, *is_training=False*)

Add elements to the Graph, computing output Tensors from input Tensors.

Subclasses must implement this method, which will be wrapped in a Template.

**Parameters**

- ***args** – Input Tensors.
- ****kwargs** – Additional Python flags controlling connection.

**Returns** output Tensor(s).

**_get_base_network_vars**()

Returns a list of all the base network's variables.

**_normalize**(*inputs*)

Normalize between -1.0 to 1.0.

**Parameters** `inputs` – A Tensor of images we want to normalize. Its shape is (1, height, width, num_channels).

**Returns**

**A Tensor of images normalized between -1 and 1.** Its shape is the same as the input.

**Return type** outputs

**`_subtract_channels`**(*inputs, means=[123.68, 116.78, 103.94]*)

Subtract channels from images.

It is common for CNNs to subtract the mean of all images from each channel. In the case of RGB images we first calculate the mean from each of the channels (Red, Green, Blue) and subtract those values for training and for inference.

**Parameters**

- **`inputs`** – A Tensor of images we want to normalize. Its shape is (1, height, width, num_channels).
- **`means`** – A Tensor of shape (num_channels,) with the means to be subtracted from each channels on the inputs.

**Returns**

**A Tensor of images normalized with the means.** Its shape is the same as the input.

**Return type** outputs

**`get_base_network_checkpoint_vars`**()

Returns the vars which the base network checkpoint will load into.

We return a dict which maps a variable name to a variable object. This is needed because the base network may be created inside a particular scope, which the checkpoint may not contain. Therefore we must map each variable to its unscoped name in order to be able to find them in the checkpoint file.

**`get_trainable_vars`**()

Returns a list of the variables that are trainable.

If a value for *fine_tune_from* is specified in the config, only the variables starting from the first that contains this string in its name will be trainable. For example, specifying *vgg_16/fc6* for a VGG16 will set only the variables in the fully connected layers to be trainable. If *fine_tune_from* is None, then all the variables will be trainable.

**Returns** a tuple of *tf.Variable*.

**Return type** trainable_variables

**class** `luminoth.models.base.`**`TruncatedBaseNetwork`**(*config*, *name='truncated_base_network'*, ***kwargs*)

Feature extractor for images using a regular CNN.

By using the notion of an "endpoint", we truncate a classification CNN at a certain layer output, and return this partial feature map to be used as a good image representation for other ML tasks.

**`_build`**(*inputs*, *is_training=False*)

**Parameters** `inputs` – A Tensor of shape *(batch_size, height, width, channels)*.

**Returns**

**A Tensor of shape** *(batch_size, feature_map_height, feature_map_width, depth)*. The resulting dimensions depend on the CNN architecture, the endpoint used, and the dimensions of the input images.

**Return type** feature_map

**`_get_endpoint`**(*endpoints*)
Returns the endpoint tensor from the list of possible endpoints.

Since we already have a dictionary with variable names we should be able to get the desired tensor directly. Unfortunately the variable names change with scope and the scope changes between TensorFlow versions. We opted to just select the tensor for which the variable name ends with the endpoint name we want (it should be just one).

**Parameters** **`endpoints`** – a dictionary with {variable_name: tensor}.

**Returns** a tensor.

**Return type** endpoint_value

**`get_trainable_vars`**()
Returns a list of the variables that are trainable.

**Returns** a tuple of *tf.Variable*.

**Return type** trainable_variables

**class** `luminoth.models.fasterrcnn.`**`FasterRCNN`**(*config*, *name='fasterrcnn'*)
Faster RCNN Network module

Builds the Faster RCNN network architecture using different submodules. Calculates the total loss of the model based on the different losses by each of the submodules.

It is also responsible for building the anchor reference which is used in graph for generating the dynamic anchors.

**`_build`**(*image*, *gt_boxes=None*, *is_training=False*)
Returns bounding boxes and classification probabilities.

**Parameters**

- **`image`** – A tensor with the image. Its shape should be *(height, width, 3)*.
- **`gt_boxes`** – A tensor with all the ground truth boxes of that image. Its shape should be *(num_gt_boxes, 5)* Where for each gt box we have (x1, y1, x2, y2, label), in that order.
- **`is_training`** – A boolean to whether or not it is used for training.

**Returns**

**A tensor with the softmax probability for** each of the bounding boxes found in the image. Its shape should be: (num_bboxes, num_categories + 1)

**classification_bbox: A tensor with the bounding boxes found.** It's shape should be: (num_bboxes, 4). For each of the bboxes we have (x1, y1, x2, y2)

**Return type** classification_prob

**`_generate_anchors`**(*feature_map_shape*)
Generate anchor for an image.

Using the feature map, the output of the pretrained network for an image, and the anchor_reference generated using the anchor config values. We generate a list of anchors.

Anchors are just fixed bounding boxes of different ratios and sizes that are uniformly generated throught the image.

**Parameters** **`feature_map_shape`** – Shape of the convolutional feature map used as input for the RPN. Should be (batch, height, width, depth).

**Returns**

**A flattened Tensor with all the anchors of shape** *(num_anchors_per_points * feature_width * feature_height, 4)* using the (x1, y1, x2, y2) convention.

**Return type** all_anchors

**get_trainable_vars**()
Get trainable vars included in the module.

**loss**(*prediction_dict*, *return_all=False*)
Compute the joint training loss for Faster RCNN.

**Parameters** **prediction_dict** – The output dictionary of the _build method from which we use two different main keys:

**rpn_prediction: A dictionary with the output Tensors from the** RPN.

**classification_prediction: A dictionary with the output Tensors** from the RCNN.

**Returns** If *return_all* is False, a tensor for the total loss. If True, a dict with all the internal losses (RPN's, RCNN's, regularization and total loss).

**summary**
Generate merged summary of all the sub-summaries used inside the Faster R-CNN network.

**class** luminoth.models.ssd.**SSD**(*config*, *name='ssd'*)
SSD: Single Shot MultiBox Detector

**_build**(*image*, *gt_boxes=None*, *is_training=False*)
Returns bounding boxes and classification probabilities.

**Parameters**

- **image** – A tensor with the image. Its shape should be *(height, width, 3)*.
- **gt_boxes** – A tensor with all the ground truth boxes of that image. Its shape should be *(num_gt_boxes, 5)* Where for each gt box we have (x1, y1, x2, y2, label), in that order.
- **is_training** – A boolean to whether or not it is used for training.

**Returns**

predictions: proposal_prediction: A dictionary with:

**proposals: The proposals of the network after appling some** filters like negative area; and NMS

proposals_label: A tensor with the label for each proposal. proposals_label_prob: A tensor with the softmax probability

for the label of each proposal.

bbox_offsets: A tensor with the predicted bbox_offsets class_scores: A tensor with the predicted classes scores

**Return type** A dictionary with the following keys

**get_trainable_vars**()
Get trainable vars included in the module.

**loss**(*prediction_dict*, *return_all=False*)
Compute the loss for SSD.

**Parameters** **prediction_dict** – The output dictionary of the _build method from which we use different main keys:

cls_pred: A dictionary with the classes classification. loc_pred: A dictionary with the localization predictions target: A dictionary with the targets for both classes and

localizations.

**Returns** A tensor for the total loss.

**`summary`**
Generate merged summary of all the sub-summaries used inside the ssd network.

## 1.11 Checkpoint management

## 1.12 Cloud management

## 1.13 Dataset management

## 1.14 Evaluating a model

## 1.15 Predict with a model

## 1.16 Web server

## 1.17 Training a model

# Index